

S

Fixed Points

CIS352 — Fall 2022

Kris Micinski



Last lecture: encoding Scheme in the lambda calculus

```
e ::= (letrec ([x (lambda (x ...) e)]))
    | (let ([x e] ...) e)
    | (lambda (x ...) e)
    | (e e ...)
    | x
    | (if e e e)
    | (prim e e) | (prim e)
    | d
d ::=  $\mathbb{N}$  | #t | #f | '()'
x ::= <vars>
prim ::= + | - | * | not | cons | ...
```

Right now: clone the corresponding autograder exercise for this lecture so you can get participation points...

Last lecture: encoding Scheme in the lambda calculus

But didn't do letrec

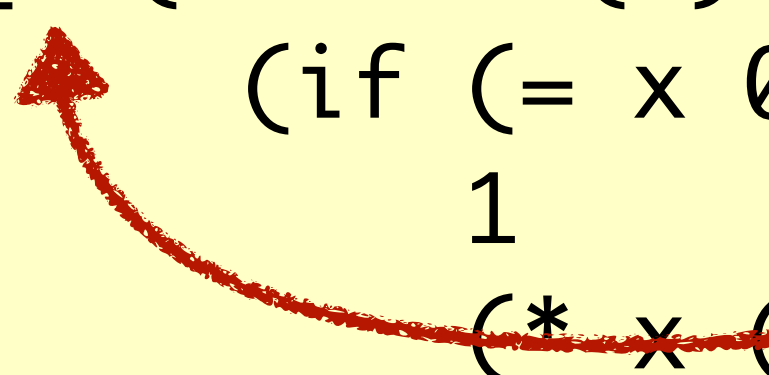
```
e ::= (letrec ([x (lambda (x ...) e)]))
    | (let ([x e] ...) e)
    | (lambda (x ...) e)
    | (e e ...)
    | x
    | (if e e e)
    | (prim e e) | (prim e)
    | d
d ::=  $\mathbb{N}$  | #t | #f | '()
x ::= <vars>
prim ::= + | - | * | not | cons | ...
```

Letrec lets us define recursive loops

```
(letrec ([f (lambda (x)
             (if (= x 0)
                 1
                 (* x (f (sub1 x))))))]
  (f 20))
```

Letrec lets us define recursive loops

```
(letrec ([f (lambda (x)
            (if (= x 0)
                1
                (* x (f (sub1 x))))))]
  (f 20))
```



Unlike **let**, letrec allows referring to f **within** its definition

Unlike **let**, letrec allows referring to f **within** its definition

```
(define (fib-using-letrec x)
  (letrec ([fib (lambda (x)
                 ;; Your answer:
                 'todo)])
    (fib x)))
```

Today, we will discuss a magic term, **Y**, that allows us to write...

```
(letrec ([f (lambda (x)
            (if (= x 0)
                1
                (* x (f (sub1 x))))))]
  (f 20))
```

```
(let ([f
      (Y (lambda (f)
          (lambda (x)
            (if (= x 0)
                1
                (* x (f (- x 1))))))]
  (f 20))
```


This magic term, named Y, allows us to construct recursive functions.

```
(define Y (λ (g) ((λ (f) (g (λ (x) ((f f) x))))  
                  (λ (f) (g (λ (x) ((f f) x)))))))
```

First, the U combinator

```
(define U (lambda (x) (x x)))
```

The U combinator lets us do something very crucial: pass a copy of a function to itself.

Let's say I didn't have letrec, what could I do...?

First observation: pass `f` to **itself**

```
(let ([f (lambda (mk-f)
          (lambda (x)
            (if (= x 0)
                1
                (* x ((mk-f mk-f) x))))))]
      ((f f) 20))
```

`mk-f` is pronounced "make f"

```
(let ([f (lambda (mk-f)
          (lambda (x)
            (if (= x 0)
                1
                (* x ((mk-f mk-f) (sub1 x))))))]
      ((f f) 20))
```

Let's see why this works!

```
(let ([f (lambda (mk-f)
          (lambda (x)
            (if (= x 0)
                1
                (* x ((mk-f mk-f) (sub1 x))))))]
      ((f f) 20))
```

Let's see why this works!

1: First, apply f to itself. First lambda goes away, returns (lambda (x) ...) with mk-f bound to mk-f

This initial call "makes the next copy"

```
(let ([f (lambda (mk-f)
          (lambda (x) ;; x = 20
            (if (= x 0)
                1
                (* x ((mk-f mk-f) (sub1 x))))))]
      ((f f) 20))
```

Let's see why this works!

1: First, apply f to itself. First lambda goes away, returns (lambda (x) ...) with mk-f bound to mk-f

2: Second, apply that (lambda (x) ...) to 20, take false branch

```
(let ([f (lambda (mk-f)
          (lambda (x)
            (if (= x 0)
                1
                (* x ((mk-f mk-f) (sub1 x))))))]
      ((f f) 20))
```

Let's see why this works!

1: First, apply `f` to itself. First `lambda` goes away, returns `(lambda (x) ...)` with `mk-f` bound to `mk-f`

2: Next, apply that `(lambda (x) ...)` to `20`, take false branch

3: Next, compute `(mk-f mk-f)`, which **gives us another copy of `(lambda (x) ...)`**

```
(let ([f (lambda (mk-f)
          (lambda (x)
            (if (= x 0)
                1
                (* x ((mk-f mk-f) (sub1 x))))))]
      ((f f) 20))
```

Let's see why this works!

1: First, apply f to itself. First lambda goes away, returns (lambda (x) ...) with mk-f bound to mk-f

2: Next, apply that (lambda (x) ...) to 20, take false branch

3: Next, compute (mk-f mk-f), which **gives us another copy of (lambda (x) ...)**

4: Apply that same function again (until base case)!

The U combinator recipe for recursion...

```
(letrec ([f (lambda (x) e-body)])  
  letrec-body)
```

Systematically translate any letrec by:

- Wrapping $(\text{lambda } (x) \text{ e-body})$ in $(\text{lambda } (f) \dots)$
- Changing occurrences of f (in e-body) to $(f f)$
- Apply U combinator / apply function to itself
- Changing letrec to let

Think carefully why this works..!

The U combinator recipe for recursion...

```
(letrec ([f (lambda (x) e-body)])  
  letrec-body)
```

Systematically translate any letrec by:

- Wrapping (lambda (x) e-body) in (lambda (f) ...)
- Changing occurrences of f (in e-body) to (f f)
- Apply U combinator / apply function to itself
- Changing letrec to let

```
(let ([f (U (lambda (f)  
            ;; replace f w/ (f f)  
            (lambda (x) e-body)))]  
  letrec-body)
```

Let's do an example...

```
(define (length-using-letrec lst)
  (letrec ([len (lambda (x)
                (if (null? x)
                    0
                    (add1 (len (rest x))))))]
    (len lst)))
```

Your job...

```
(define (length-using-u lst)
  (let ([len (U (lambda (f)
                 (lambda (x)
                   'todo)))]
        (len lst)))
```

Now another example...

```
(define (fib-using-letrec n)
  (letrec ([fib
            (lambda (x)
              (cond [(= x 0) 1]
                    [(= x 1) 1]
                    [else (+ (fib (- x 1))
                              (fib (- x 2)))]))]
            (fib n)))
```

Translate **this** one to use U

```
(define (fib-using-U n)
  (letrec ([fib (U 'todo)])
    (fib n)))
```

```
(let ([f (lambda (mk-f)
          (lambda (x)
            (if (= x 0)
                1
                (* x ((mk-f mk-f) (sub1 x))))))]
      ((U f) 20))
```

One pesky thing: need to rewrite function so that calls to mk-f need to first “get another copy” by doing (mk-f mk-f)

By contrast, the **Y** combinator will allow us to write **this**

```
(let ([f (lambda (f)
          (lambda (x)
            (if (= x 0)
                1
                (* x (f (sub1 x))))))]
      ((Y f) 20))
```

```
(let ([f (Y (lambda (f)
             ;; no change to e-body
             (lambda (x) e-body)))]
      letrec-body)
```

Let's ask ourselves: what does f need to **be** when Y plugs it in...?

$$(Y f) = f (Y f)$$

Deriving Y

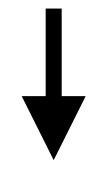
$$(Y f) = (f (Y f))$$

$$Y = (\lambda (f) (f (Y f))) \quad 1. \text{ Treat as definition}$$

$$mY = (\lambda (mY) (\lambda (f) (f ((mY mY) f)))) \quad \begin{array}{l} 2. \text{ Lift to } mY, \\ \text{use self-application} \end{array}$$

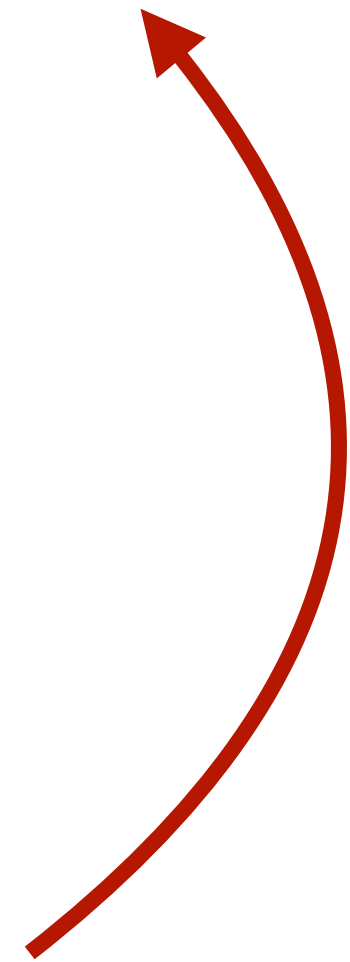
$$mY = (\lambda (mY) (\lambda (f) (f (\lambda (x) (((mY mY) f) x)))))) \quad 3. \text{ Eta-expand}$$

U-combinator: (U U) is Omega



$$Y = (U (\lambda (y) (\lambda (f) (f (\lambda (x) (((y y) f) x))))))$$

$$mY = (\lambda (mY) (\lambda (f) (f (\lambda (x) (((mY mY) f) x))))))$$



$$(Y f) = f (Y f)$$

By contrast, the **Y** combinator will allow us to write **this**

```
(let ([f (lambda (f)
          (lambda (x)
            (if (= x 0)
                1
                (* x (f (sub1 x))))))]
      ((Y f) 20))
```

Closing words of advice:

- Understand how to write recursive functions w/ U / Y
- Do not need to remember precisely why Y works
 - But do need to remember how to use it!
- If you want to understand: just think carefully about what U / Y are doing (with examples)